Software Guide

For

Generic Sensor Hub

Powered by icm-20690
# TABLE OF CONTENTS

TABLE OF CONTENTS .................................................................................................................. 2
LIST OF FIGURES ......................................................................................................................... 3
LIST OF TABLES ............................................................................................................................ 3
Useful Links .................................................................................................................................. 4
References ...................................................................................................................................... 4
Acronyms ....................................................................................................................................... 4

1 Overview ..................................................................................................................................... 5
  1.1 Introduction .......................................................................................................................... 5
  1.2 Definitions ............................................................................................................................ 5
    1.2.1 Sensor (aka primary sensor) ......................................................................................... 5
    1.2.2 Auxiliary sensor .......................................................................................................... 5
    1.2.3 VSensor ....................................................................................................................... 5
    1.2.4 Custom sensor ............................................................................................................. 5
    1.2.5 Host ............................................................................................................................. 6

2 Software architecture .................................................................................................................. 6
  2.1 Overview .............................................................................................................................. 6
  2.2 VSensor framework .............................................................................................................. 6
    2.2.1 VSensor ....................................................................................................................... 7
    2.2.2 VSensorListener ........................................................................................................... 7
    2.2.3 VSensorEvents ............................................................................................................ 7
    2.2.4 VSensorData ............................................................................................................... 8
    2.2.5 Generic Sensor Hub VSensor flow .............................................................................. 8

2.3 Extension Framework ............................................................................................................. 10
  2.3.1 Custom Sensor ................................................................................................................ 10
  2.3.2 Custom Sensor Listener .................................................................................................. 11
  2.3.3 Extended Framework initialization ................................................................................ 11
  2.3.4 Activation of GSH custom sensor examples .................................................................. 12

2.4 FrontEnd ................................................................................................................................. 12
  2.4.1 Introduction to decimation ............................................................................................. 12
  2.4.2 User application task mode ........................................................................................... 15
  2.4.3 void InvGshFrontEnd_init(void) .................................................................................. 15
  2.4.4 int InvGshFrontEnd_initListeners(void) ..................................................................... 15
  2.4.5 int InvGshFrontEnd_acknowledgeReset(void) ............................................................. 15
  2.4.6 int GshFrontEnd_handleCommandWrapper (enum DynProtocolEid eid, const DynProtocolEdata_t * edata, DynProtocolEdata_t * respdata) .......................................................... 15

2.5 GSH Configuration module .................................................................................................. 16
  2.5.1 Configuration macros ................................................................................................. 16
  2.5.2 GshConfig public API ............................................................................................... 16
  2.5.3 GshConfig private API .............................................................................................. 17
  2.5.4 GshConfig callbacks for IRQ and hardware sensors .................................................... 18
LIST OF FIGURES

Figure 1 : Generic Sensor Hub software architecture overview .................................................................. 7
Figure 2 : VSensor flow instantiated by GSH .......................................................................................... 9
Figure 3 : GshFrontEnd class diagram .................................................................................................. 14
Figure 4 : GSH main function sequence diagram .................................................................................. 20
Figure 5 : FreeRTOS tasks used in GSH ................................................................................................. 23

LIST OF TABLES

Table 1 : Example of “standardized” VSensorData types ........................................................................ 8
USEFUL LINKS

InvenSense website:
http://www.InvenSense.com/

STNucleo website:

FreeRtos:
http://www.freertos.org/

REFERENCES

[3] – Invensense GSH doxygen – Invensense (<GSH-sdk>\invn\firmware\gsh\sdk\nucleo\win32\<version>\doc\html\index.html)

ACRONYMS

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODR</td>
<td>Output data rate</td>
</tr>
<tr>
<td>RI</td>
<td>Report interval</td>
</tr>
<tr>
<td>SHExt</td>
<td>Sensor hub extension framework</td>
</tr>
<tr>
<td>VSensor</td>
<td>Virtual sensor</td>
</tr>
</tbody>
</table>
1 OVERVIEW

The purpose of this document is to describe Invensense Generic Sensor Hub (GSH) software.

This document contains:
- details on the content of the GSH package
- description of the architecture of the GSH software
- description of the most common GSH use cases

This document does not contain:
- A quick start guide. See [1] for such information.
- Information about hardware reference kit. See [1] for such information.
- User guide information on third party software and tools such as freertos, STM32 drivers, linaro... Please refer to relevant websites for detailed information on these topics.

1.1 INTRODUCTION

The purpose of GSH is to have a complete sensor hub solution portable on various MCU. The code is provided as open-source with motion algorithms only delivered as libraries. The proposed implementation supports the ST Nucleo board (embedding a STM32F411RE) with an ICM-20690 MEMS sensor connected. The ICM-20690 MEMS is a 6-axis combo (accelerometer and gyroscope).

1.2 DEFINITIONS

1.2.1 Sensor (aka primary sensor)

In this document a sensor always refers to a hardware device outputting numerical data resulting from physical measures. Such a device also known as primary sensor is connected to the main MCU.

GSH hardware reference kit contains a nucleo carrier board allowing to plug up to 3 sensors. In GSH hardware reference kit:
- motion sensor is Invensense ICM20690
- magnetometer is AKM09911
- pressure/temperature sensor is BMP280
- light/proximity sensor is VCNL4040

Note: AKM09911, BMP280 and VCNL4040 are embedded on the same sensor daughter board.

1.2.2 Auxiliary sensor

An auxiliary sensor is an additional physical sensor connected as a slave of the ICM sensor. Thus such a sensor is not accessible directly from MCU but only through the ICM.

1.2.3 VSensor

A VSensor is a virtual sensor satisfying Invensense VSensor framework definition (see §2.2 for details)

A VSensor is a software component which:
- listens data at programmable frequency from one or more sensor and/or VSensor
- possibly applies some algorithms on collected data
- signals new available data to its listeners

1.2.4 Custom sensor

A custom sensor is software component satisfying Invensense Extension framework definition. The Extension Framework is a simplified version of the VSensor framework initially designed to be compliant with Invensense Sensor Studio. It allows user to create its own virtual sensor running in a separate task in the system.
1.2.5 Host

Host is a device may it be a computer or an application processor, that sends commands to the GSH and listens to its published data.

By default, GSH does not start any sensor at boot time. It just waits for commands on its link to host. GSH currently supports UART only to connect to host.

Note: a particular mode allows to compile a user task inside GSH firmware (see §4.5). Such a task is indeed an RTOS thread dedicated to run user application code. Some mechanisms are available to allow user task to send command to GSH framework task and receive sensor data. In that case the user task behaves as a host as defined in this chapter.

2 SOFTWARE ARCHITECTURE

2.1 OVERVIEW

Generic Sensor Hub software architecture is divided in 3 core modules:

- VSensor framework is in charge of listening data from one or more Sensor/VSensor, processing data using various algorithms and notifying its listeners of new data availability.
  
  VSensor Framework is described in §2.2

- Extended framework brings the possibility for user to add custom sensors running in an isolated and simplified framework.
  
  Extended Framework is described in §2.3

- GSH frontend is the interface between VSensor framework and remote control application. It is based on GshFrontEnd module in charge of:
  
  o instantiating V Sensors and listening to their data
  o reading/answering commands and sending V Sensor data outside GSH through Dynamic Protocol Adapter.

  GSH frontend is described in §2.4.

Those three core modules use several side modules:

- GshConfig module contains hardware initialization functions, RTOS initialization and tasks implementation and implementation of some hooks required by other modules. It also contains all hooks and interrupt handlers for all physical sensors.
  
  GshConfig aims at gathering all pieces of software that one should update to add new use cases to GSH (see §2.5).

- Main module contains main function and is in charge of booting the system (See §2.6)

- Dynamic protocol adapter is a module handling communication between GSH and outside world (see §2.7)

- Icm20690 device driver abstract control of sensor ICM20690

- board_hal offers an API to abstract control of hardware platform. In GSH reference design, the hardware platform is a STM nucleo board STM32F411RE whose driver is also embedded in GSH software.

2.2 VSENSOR FRAMEWORK

VSensor framework is stored in <gsh_root_dir>/sources/Invn/Vsensor

This directory contains the modules that allows to manage V Sensors in a generic manner. But it does not contain any V Sensor implementation, just base class definition.

Figure 2 shows the class diagrams of all V Sensors implemented in Generic Sensor Hub.

Next chapters give the definition of VSensor and other base concepts of VSensor framework. For full description of attributes and methods please refer to doxygen documentation [3].

Note: implementations of V Sensors used by Generic Sensor Hub are located in in <gsh_root_dir>/sources/GSH/VSensorImpl.
2.2.1 VSensor

A VSensor can publish data coming from a HW resource (HW sensors) or from another VSensor. Thus VSensor’s implementation can instantiate its own set of VSensorListeners to retrieve input data from other VSensors for its own purpose.

Moreover, a VSensor maintains a list of subscribers that are to be notified when the VSensor produces some new data.

2.2.2 VSensorListener

VSensorListener, standing for Virtual Sensor Listener, is the object that allows controlling of a VSensor and receiving data from it. In Generic Sensor Hub VSensorListener objects are used by some VSensors to retrieve data from other VSensors. They are also used by GshFrontEnd to retrieve and output data from all VSensors (see Figure 3 and Figure 2).

Data and other events will be received by means of an event callback, in which the VSensorListener can process it.

2.2.3 VSensorEvents

VSensors communicate each other using events called VSensorEvents.

For example a VSensor sends VSENSOR_EVENT_NEW_DATA event, by means of VSensor_notifyEvent() function to notify each of its listeners of new data. This event must be associated with the data value and the data timestamp.

See [33] for description of all events supported by VSensors.
2.2.4 VSensorData
Primary purpose of a VSensor is to publish data to a set of listeners.
Each data notification must be associated with a timestamp; supposedly corresponding to the time of the data was generated.

A VSensor can publish data of any arbitrary type (by means of the generic pointer void *) with the obvious implication that listener’s must be aware of this type to handle it properly.

However, some “standardized” data types are defined to allow listeners to register to a VSensor disregarding its actual implementation, as long as it publishes expected data type. It allows creation of more generic module (VSensor, Frontend ...).

A non-exhaustive list of such “standard” data is given in following table (see [33] for all details):

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Format</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSensorDataAccelerometer</td>
<td>x, y, z accel data</td>
<td>Fixp Q16</td>
<td>g</td>
</tr>
<tr>
<td>VSensorDataGyrometer</td>
<td>x, y, z gyro data</td>
<td>Fixp Q16</td>
<td>deg/s</td>
</tr>
<tr>
<td>VSensorDataMagnetometer</td>
<td>x, y, z compass data</td>
<td>Fixp Q16</td>
<td>µT</td>
</tr>
<tr>
<td>VSensorDataQuaternion</td>
<td>w, x, y, z + accuracy</td>
<td>Fixp Q30 + Q16</td>
<td>Accuracy in deg</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Example of “standardized” VSensorData types

2.2.5 Generic Sensor Hub VSensor flow
Figure 2 shows the VSensor flow instantiated by GSH as implemented in GSH sdk.
Figure 2: VSensor flow instantiated by GSH
2.3 EXTENSION FRAMEWORK

Extension framework also known as Sensor Hub extension framework or SHExt allows to easily add custom sensors to the Generic Sensor Hub. A Custom Sensor is a virtual sensor which inherits the properties of VSensors. Thus a custom sensor may listen to other virtual sensors (custom or VSensor one) and may publish data to its listeners. In addition, custom sensor offers the possibility to be run in a separated RTOS thread enabling it to generate data periodically on its own and limiting the impact of possible side effects while adding custom features. Moreover, a custom sensor maybe automatically generated by Invensense Sensor Studio.

GSH extension framework allows to add a maximum of 8 custom sensors.

For full description of custom sensor types, attributes and methods, refer to [3].

As an example, GSH sdk proposes implementation of four custom sensors in <gsh_root_dir>/sources/GSH/ExtFwk/examples.

2.3.1 Custom Sensor

A Custom Sensor or custom VSensor is an object of type inv_shext_vsensor_t which is mostly defined by an event handler function of type inv_shext_vsensor_event_cb, that is called by the system when needed. A Custom VSensor implementation must then implement this function and handle the following events: INIT, SUBSCRIBE, UNSUBSCRIBE, UPDATE_PERIOD, SET_CONFIG, GET_CONFIG (see [3]: inv_shext_vsensor_event).

Example of custom sensor event handler taken from <gsh_root_dir>/sources/GSH/ExtFwk/examples/CustomSensor0.c:

```c
/* This function corresponds to the main event handler for a sensor.  
 * It acts as a dispatcher to the functions above in order to process  
 * each events (or actions).  
 */
static int MyCustomSensor0_eventCb(struct inv_shext_vsensor * sensor,  
                           enum inv_shext_vsensor_event event, const void * data, uint16_t size)
{
    (void)sensor;
    int rc = 0;

    switch(event) {
    case INV_SHEXT_VSENSOR_EVENT_INIT:  
        rc = MyCustomSensor0_onInit();  
        break;
    case INV_SHEXT_VSENSOR_EVENT_SUBSCRIBE:  
        rc = MyCustomSensor0_onSubscribe();  
        break;
    case INV_SHEXT_VSENSOR_EVENT_UNSUBSCRIBE:  
        rc = MyCustomSensor0_onUnsubscribe();  
        break;
    case INV_SHEXT_VSENSOR_EVENT_UPDATE_PERIOD:  
        //convert data from us to ms to keep compatibility  
        rc = MyCustomSensor0_onUpdatePeriod((*(const uint32_t *)data)/1000);  
        break;
    case INV_SHEXT_VSENSOR_EVENT_SET_CONFIG:  
        rc = MyCustomSensor0_onSetConfig(data, size);  
        break;
    case INV_SHEXT_VSENSOR_EVENT_GET_CONFIG:  
        rc = MyCustomSensor0_onGetConfig((void *)data, size);  
        break;
    default:
        rc = -1;
        break;
    }

    return rc;
}
```
To generate data periodically on its own, the Custom VSensor implementation must use task objects of type inv_shext_task. Then inv_shext_task API allows to control task start, stop and period. See custom sensor 0 for an example in <gsh_root_dir>/sources/GSH/ExtFwk/examples/CustomSensor0.c.

An additional public function is needed to act as a constructor called by inv_shext_entry function at boot time (see §2.3.3) as shown in next example.

```c
void MyCustomSensor1_entry(void)
{
    /* initialize custom sensor object and register it to the system */
    inv_shext_vsensor_register(&MyCustomSensor1,
                              MyCustomSensor1_eventCb, VSENSOR_TYPE_CUSTOM(1));
}
```

Note that inv_shext_vsensor_register will create sensor object and register it to FrontEnd as well.

### 2.3.2 Custom Sensor listener

A custom Sensor listener objects allows to receive data event from other custom sensor and/or VSensor registered to the system.

A listener is linked to (or attached to) a VSensor and must handle events: NEW_DATA, PERIOD_UPDATED sent by the VSensor (see [3]: inv_shext_vsensor_listener_attach method for details). The attachment should be done by custom sensor event callback when it receives INIT event from the system.

Example of Custom Sensor listening accelerometer:

```c
static int MyCustomSensor_onInit(void)
{
    if(inv_shext_vsensor_listener_attach(&MyAccListener,
                                         VSENSOR_TYPE_ACCELEROMETER, &MyCustomSensor,
                                         MyAccListener_eventCb)) {
        return -1;
    }
    return 0;
}
```

### 2.3.3 Extended Framework initialization

At boot time GSH main calls two functions that are related to extension framework. First one, inv_shext_init, executes software initialization of the extension framework while second one, inv_shext_user_entry, is in charge of creating the custom sensors. The latter one is just a wrapper around a weak function named inv_shext_entry which is expected to be overloaded by user in order to implement the creation of his own custom sensors.

In custom sensors example provided in GSH sdk, implementation of inv_shext_entry which calls the four custom sensors’ entry functions may be found in <gsh_root_dir>/sources/GSH/ExtFwk/examples/CustomEntry.c.

```c
void inv_shext_entry(void)
{
    MyCustomSensor0_entry();
    MyCustomSensor1_entry();
    MyCustomSensor2_entry();
    MyCustomSensor3_entry();
}
```
Each custom sensor should implement its own custom sensor entry function. It is generally the only public function in a custom sensor implementation. It should at least call inv_shext_vsensor_register to create custom sensor object and register it to FrontEnd (see [3] for details on inv_shext_vsensor_register).

See §4.2 for more information about adding a custom sensor.

### 2.3.4 Activation of GSH custom sensor examples

Four custom sensors example are provided with GSH sdk. They are compiled out by default. To activate them just add the custom entry and custom sensor source and header files to the list of source files in makefile.

In `<gsh_root_dir>/config/linaro-cm4-nucleo/main.mk` add the bold lines in following code extract:

```markdown
DEPS    +=  
    sources/GSH/ExtFwk/examples/Ak09911Driver.h  
    sources/board-hal/common.h  
    sources/board-hal/dbg_gpio.h  
    sources/board-hal/delay.h  
    (...)  
CSRCS   +=  
    sources/GSH/ExtFwk/examples/CustomEntry.c  
    sources/GSH/ExtFwk/examples/CustomSensor0.c  
    sources/GSH/ExtFwk/examples/CustomSensor1.c  
    sources/GSH/ExtFwk/examples/CustomSensor2.c  
    sources/GSH/ExtFwk/examples/CustomSensor3.c  
    sources/GSH/ExtFwk/examples/Ak09911Driver.c  
    sources/board-hal/common.c  
    sources/board-hal/dbg_gpio.c  
    sources/board-hal/delay.c  
    (...)  
```

### 2.4 FRONTEND

Frontend is implemented in `<gsh_root_dir>/sources/GSH/FrontEnd/`.

Frontend module is responsible for bidirectional communication with outside world. On input side FrontEnd reads, handles and answers incoming commands. On output side FrontEnd is in charge of listening all running VSensor and outputting their data to host at the requested RI.

The actual communication (meaning frames encode/decode send/receive) is deported to a dedicated communication protocol module described in §2.6.

FrontEnd exposes a public API briefly described in following chapters.

Figure 3 presents the class diagram of GshFrontEnd as implemented in GSH.

### 2.4.1 Introduction to decimation

Different RI requested for the various VSensor may lead FrontEnd to drop some of the samples. Decimation is used at frontend level to decide if data currently received from a VSensor should be either output to host as is or is dropped.

To explain why decimation is needed, consider the following simple scenario:

- User sends from Host a command to start a VSensor V0 with a reporting interval (a.k.a. RI) of 5ms
- V0 listens to another VSensor V1. Thus framework will also start V1 with RI=5ms. But Frontend will listen and output only V0 data every 5ms.
- User now sends a command to start VSensor V1 with RI=10ms. But V1 is already reporting data every 5ms as requested by V0. So framework keeps V1 RI=5ms.
- Frontend now listens to V0 and V1. But V1 notifies new data every 5ms whereas Frontend should output V1 data every 10ms only. In such a case a decimation algorithm should drop every second sample from V1.

GshFrontEnd offers possibility to choose a listener for each VSensor among two different types. Each listener offers different capabilities in terms of decimation listed hereafter:

- INV_GSH_FRONTEND_DECIMATOR_NONE: No decimation. Data are reported at the actual rate of the VSensor
- INV_GSH_FRONTEND_DECIMATOR_SIMPLE: Drop current data if less than 75% of expected reporting interval has passed since last data.
Figure 3: GshFrontEnd class diagram
2.4.2 User application task mode

GSH can be used to build an autonomous system meaning that host is no more a remote machine or application processor. Host is the MCU running GSH framework itself. In that case a dedicated thread is created at boot time in order to isolate application from GSH framework.

An example of such a task exists in GSH sdk. It may be activated at compilation time using a dedicated macro in GSH configuration module (see §2.5.1 for details on this macro). In this case user task is considered being an additional host for GSH and some mechanisms are put in place to allow bi-directional communication between user application task and framework task.

To sum up, framework message queue is used by user application task to send commands while framework task sends sensors data in a dedicated message queue to user application task.

See §2.8.2 for more details on FreeRtos tasks.

2.4.3 void InvGshFrontEnd_init(void)

This function initializes some software modules (no hardware initialization done here). In particular, it instantiates and initializes:

- the transport layer of the communication protocol. It registers a transport event handler function named GshFrontEnd_transportEvent_cb().
- the base layer of the communication protocol. It registers a protocol event handler function named GshFrontEnd_protocolEvent_cb()

It also enables DMA for transport layer on output side.

```c
void InvGshFrontEnd_init(void)
{
    memset(&sGshFrontEnd, 0, sizeof(sGshFrontEnd));
    DynProTransportUart_init(&sGshFrontEnd.transport, GshFrontEnd_transportEvent_cb, 0);
    DynProTransportUart_enableTxDma(&sGshFrontEnd.transport);
    DynProtocol_init(&sGshFrontEnd.protocol, GshFrontEnd_protocolEvent_cb, 0);
}
```

2.4.4 int InvGshFrontEnd_initListeners(void)

This function creates and attaches a listener to each VSensor registered at front end level. Note that listeners are not enabled by default. They are enabled when GshFrontEnd receives relevant command from host.

2.4.5 int InvGshFrontEnd_acknowledgeReset(void)

When GSH receives a reset command it executes a reset procedure whose implementation depends on the board running GSH (see §2.5). Once reset is executed and boot procedure is successful GSH should warn host that it is ready to receive some commands. For that purpose, InvGshFrontEnd_acknowledgeReset function sends a “response to reset” frame over protocol just before starting FreeRtos scheduler.

Note: this frame is also sent after a power on reset, whereas no reset command was sent by host. This frame may anyway be used by host to detect GSH successfully booted.

2.4.6 int GshFrontEnd_handleCommandWrapper (enum DynProtocolEid eid, const DynProtocolEdata_t * edata, DynProtocolEdata_t * respdata)

This function allows to request command handling from outside Gsh FrontEnd. This is typically used to call framework commands from user application task (see §2.4.2).
See more details on the function parameters in [3].

### 2.5 GSH CONFIGURATION MODULE

GSH configuration module is stored in `<gsh_root_dir>/sources/GSH/GshConfig.*`. This module covers following functionalities:
- boot and initialization of the system
- initialization of the RTOS and implementation of the threads
- implementation of several hooks required by some other modules
- implementation of hooks and callback needed to support hardware sensors AKM09911, BMP280 and VCNL4040.

This module aims gathering as much as possible the software that user need to add new functionalities. For example, adding another hardware sensor, moving to another RTOS.

Content of the module is described in next paragraphs. For more detailed information see [3].

#### 2.5.1 Configuration macros

Few configuration macros are defined at the beginning of GshConfig.c. Note that these macros may be tightly linked to hardware and must be handled carefully when GSH is moved to another platform.

- **USE_SPI_TO_ACCESS_MOTION_SENSOR**
  Select communication link between STNucleo and motion sensor. Set to 1 to use SPI, 0 to use I2C

- **AUXSENSOR_SAMPLING_TIMER**
  Select which timer should be used for auxiliary sensors. In current GSH configuration running on Nucleo-F411RE TIMER4 is used.

- **USE_USER_APP_THREAD**
  Activate the use of user_app thread if set. Desactivate if reset

- **CONFIGURE_DECIMATOR**
  Define which decimator should be used at FrontEnd level for VSensors that supports decimation. VSensor which does not support decimation are forced to `INV_GSH_FRONTEND_DECIMATOR_NONE`. See §2.4.1 for more details on decimation.

  - `INV_GSH_FRONTEND_DECIMATOR_NONE`: don't use any decimator
  - `INV_GSH_FRONTEND_DECIMATOR_SIMPLE`: use simple decimator

#### 2.5.2 GshConfig public API

```c
void GshConfig_initHardware(void)
```

This function is called once at the very beginning of main().

It should contain all hardware initialization. For example, UART, SPI, I2C, Timers, hw sensors.

```c
void GshConfig_initTimers(void)
```

This function initializes the timers needed for GSH. Some of them may cause issue if they are initialized before SPI link init on nucleo. That's why timers are initialized apart from rest of the hardware in order to be called later in main.
void GshConfig_initVSensorFlow(void)

This function is in charge of creating the VSensor graph and setting configuration values of the VSensors. It is the right place to add or remove some VSensors. The function is called once at boot time.

The VSensors graph is designed statically in this function. This means that all VSensors are created at boot time by the GSH and there is no dynamic mean to add VSensor at run time.

It is executed in 3 steps:

- **Initialize VSensors**

  First, the function initializes raw VSensors meaning VSensors who listen to hardware sensors and not VSensors. Then for each VSensor, the function calls relevant init function eventually passing in argument the VSensor(s) that will be listened to by the current VSensor.

  For example, the following function call (taken from GshConfig.c) will call Calibrated Accelerometer VSensor init function requesting it to listen to ICM20690 raw accelerometer data.

  ```c
  VSensorImplCModelCalAcc_init(VSensorImplIcm20690_getRawAccHandle());
  ```

  **Note:** if a VSensor listens to some other VSensors it should be initialized after the VSensors it listens to. In the previous example, VSensorImplCModelCalAcc_init() should be called after VSensorImplIcm20690_initRawAcc().

- **Configure VSensors**

  If relevant, VSensors configuration is done by calling VSensor_SetConfig function as show in next code extract taken from GshFrontEnd.c. Next code extract passes mounting matrix to be used for Raw accelerometer.

  ```c
  VSensor_setConfig(VSensorImplIcm20690_getRawAccHandle(), &acc_gyr_mmatrix);
  ```

- **Registers VSensors to FrontEnd**

  Last step consists in registering each VSensors information needed at frontend level in a static tab, namely pointer on VSensor object, sensor type and decimator type.

  ```c
  for(i = 0; i < sizeof(idata)/sizeof(idata[0]); ++i) {
    if(InvGshFrontEnd_registerVSensorToFrontend(idata[i].vsensor,
                                               idata[i].gshSensorType,
                                               idata[i].decimator)) {
      INV_MSG(INV_MSG_LEVEL_ERROR, "Fail to register VSensor (type=%d) to frontend with ID=%d", VSensor_getType(idata[i].vsensor), idata[i].gshSensorType);
    }
  }
  ```

int GshConfig_initRtos(void)

This function creates RTOS objects needed for the GSH. See more details in §2.8.

### 2.5.3 GshConfig private API

void fwk_thread(void * arg)

This function is the main FreeRtos thread. It runs the whole framework, handles commands and processes VSensors events. See more details in §2.8.
void user_app_thread(void * arg)

This function is only compiled when user task is activated (i.e. USE_USER_APP_THREAD is set to 1. See §2.4.2). It is a basic example of user application task. See more details in §2.8.

void fwk_RcvQueueCmdProcess(struct frameworkEvent event)

This function is only compiled when user task is activated (i.e. USE_USER_APP_THREAD is set to 1. See §2.4.2). It is used by framework task to execute command requests sent by user app task.

void vApplicationIdleHook(void)

Idle thread source code. Run only when all other threads are in blocked state.

void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName)

FreeRTOS hook called when FreeRTOS detects a stack overflow.

void InvGshFrontEnd_executeSystemResetHook(void)

This function is called in case of software reset, typically when receiving reset, setup or cleanup commands. It is hardware dependent and so may be updated in case user moves to a different hardware other than nucleo STM32F411RE.

void InvGshFrontEnd_FillQueueHook(int id, VSensorDataAny data)

Hook called by GSH FrontEnd each time one of its listener receives some new data. User may add his own code in this hook to make FrontEnd handle VSensor data differently. Typically, it is used to send VSensor data to the user application task through a message queue (this is the implementation proposed in GSH sdk under USE_USER_APP_THREAD compilation flag. See §2.4.2).

2.5.4 GshConfig callbacks for IRQ and hardware sensors

Some IRQ handlers of the board HAL offers possibility to call user defined function (aka callback). This allows user to add some code in IRQ handler without modifying hardware control layer. This eases the porting of GSH to a different RTOS.

On the other hand, primary sensors drivers often require some functions to be defined externally. Indeed, hardware sensor may not have any internal timer mechanism nor interrupt signal sent when a new data is ready. In that case a timer of the board (or timer channel in case of nucleo) is used to periodically read hardware sensors data. Thus VSensor implementation requires some time control functions to be defined externally. Using external timers exposed by board hal may also require to implement some timer IRQ callback typically in order to send some message in framework queue.

Those functions and callbacks are briefly described here after.

static void ext_interrupt_cb(void * context, int int_num)

This function is a callback that will be executed each time the motion sensor interrupt toggles. As it will be executed under interrupt mode, it should be as fast as possible. That’s why this function just sends a message containing specific event value to the FreeRTOS queue that is monitored by framework task. See more details about tasks definition in §2.8.

static void ext_interrupt_main_uart_cb(void * context)

This function is a callback called from MAIN_UART interrupt handler which is used to communicate with host. As it will be executed under interrupt mode, it should be as fast as possible. This function just sends a message containing specific event value to the FreeRTOS queue that is monitored by framework task.


```c
uint64_t <hardware_sensor>_get_time_us(void)
uint64_t <hardware_sensor>_start_timer_us(void)
uint64_t <hardware_sensor>_stop_timer_us(void)
uint64_t <hardware_sensor>_reconfigure_timer_us(void)
```

These functions offer timing functionality in case some additional hardware sensors need it. In GSH sdk it is the case for ak09911, bmp280, vncl4040 light and proximity sensors.

```c
static void ext_interrupt_start_<hardware_sensor>_cb(void *context)
```

These functions are the callbacks called from timer interrupt handlers that is used by the related hardware sensor. These functions are used to get a timestamp, store it in a ringbuffer dedicated to the hardware sensor and post a message in framework queue.

### 2.6 MAIN

It is implemented in `<gsh_root_dir>/sources/GSH/main.c`

This file is the entry point of Generic Sensor Hub. The sequence diagram of main function is described on Figure 4.
2.7 DYNAMIC PROTOCOL ADAPTER

Dynamic Protocol Adapter is the software module in charge of input and output communication between GSH and outside world. A full description of the protocol may be found in [2].

Briefly, dynamic protocol adapter is a bidirectional 2-layers protocol divided in data layer and transport layer.

2.7.1 data layer

Data layer is implemented in:

This layer is in charge of decoding input frames and encoding output frames.
From embedded system point of view, data layer decodes input frames into objects named DynProtocolEdata. Such object contains command id and parameters (if relevant). This object is then passed to the DynProtocolEvent_cb callback that will handle it.

**Note:** a callback mechanism is used here as the protocol is not expected to know how to deal with input commands. GshFrontEnd knows. That is why the DynProtocolEvent_cb callback is implemented in GshFrontEnd and registered to protocol during GshFrontEnd initialization (see §2.4.2).

On output side data layer encodes DynProtocolEdata objects into output frames. This includes response to commands as well as sensor data output.

### 2.7.2 Transport layer

Transport layer is implemented in following files:


Transport layer is in charge of accessing the physical communication link. GSH only supports UART as a physical communication link with outside world.

On input side, transport layer reads data flow on physical link, isolates frames, removes transport header and passes such cleaned frame to data layer.

On output side transport layer receives a data layer frame, encapsulates it and sends it over physical link.

### 2.8 FREERTOS CONFIGURATION

#### 2.8.1 Configuration details

FreeRTOS source code is not provided inside GSH sdk. It is up to user to download and install it in GSH. See GSH Quick Start Guide [1] for the detailed installation procedure.

**Once installed FreeRtos source code is stored in:**

`<gsh_root_dir>/sources/FreeRTOS`

This is the source code of FreeRtos as downloaded from official website (see Useful Links), not expected to be modified.

**FreeRtos configuration file is stored in:**

`<gsh_root_dir>/sources/GSH/FreeRTOS/FreeRTOSConfig.h`

User should read this file to have the whole details. Nevertheless, some important settings are listed here:

- FreeRtos is used in preemptive mode
- SystemCoreClock (HCLK) of the nucleo defines the CPU_CLOCK for FreeRtos
- Total Heap size is 14kB
- Check for stack overflow enabled. Corresponding hook is in GshConfig module (see §2.5.3: vApplicationStackOverflowHook).

#### 2.8.2 FreeRtos tasks

By default, only one FreeRtos task is created at boot time to run the whole Gsh framework including FrontEnd, VSensor framework and Extension framework. An optional task may be added to run some application code.

Note also that each custom sensor may also run in dedicated task (see §2.3).
Framework task:
Task is implemented in <gsh_root_dir>/sources/GSH/GshConfig.c:fwk_thread(). It has priority RTOS_HIGHERPRIORITY_TASK and 512 words of stack.

This task is a simple message pump meaning that it blocks on a unique message queue waiting for some events to happen. This mechanism has the advantage to serialize incoming events. It avoids blocking on several RTOS objects and doing this it prevents any deadlock or concurrent access issue.

Moreover, all hardware resources used through the board hal (timers, uart and dma) have the same interrupt priority level. This means that every event handler is run to completion (only interrupted to store some popping events in the message queue). Consequence is that each event handling must be as fast as possible to prevent any side effect on system reactivity and data latency. User should be keep this requirement in mind when adding new VSensors or custom Sensors.

This configuration also has the advantages to run framework task only when some event need to be handled which leaves more room for later power management improvements.

Note that all software entities feeding the framework message queue are located in <gsh_root_dir>/sources/GSH/GshConfig.c.

User_app_task:
Task is implemented in <gsh_root_dir>/sources/GSH/GshConfig.c:user_app_thread(). It has priority RTOS_HIGHERPRIORITY_TASK-1 and 512 words of stack.

This task is given as an example of communication between an application task and the framework task. It just sends commands to framework in order to start 2 VSensors and it then blocks on a dedicated message queue waiting for sensors data.

Two FrontEnd functions allow to support communication between framework task and user application task: GshFrontEnd_handleCommandWrapper and InvGshFrontEnd_FillQueueHook.

- GshFrontEnd_handleCommandWrapper allows to pass commands directly to the FrontEnd shunting Dynamic Protocol. This allows framework task to handle commands received from user application task.
- InvGshFrontEnd_FillQueueHook is a hook in GshFrontEnd called from GshFrontEnd_genericEventHandler but declared as extern (this avoids user to modify GshFrontEnd.c file). In GSH sdk, an example of implementation of this function may be found in GshConfig.c. It sends every sensor data to the user application message queue.

Figure 5 presents a simplified view of freeRTOS tasks used in GSH.
Figure 5: FreeRTOS tasks used in GSH

```
not fwkQueue empty?

switch(event) {
    case UART_RX_EVENT:
        DynProTransportUart_rxProcessByte();
        break;

    case SENSOR_EVENT:
        VSensorImplIcm20690_poll(); // calls GshFrontEnd
            // GshFrontEnd

    case USER_APP_EVENT:
        GshFrontEnd_handleCommandWrapper(event);
        break;
    ...
}
```

InvGshFrontEnd_FillQueueHook()

// function called by FrontEnd each time a new data is published by some VSensor
sendMessageToQueue(usrAppQueue, SENSOR_EVENT)
3 SOFTWARE PACKAGE

This chapter gives a macro view of the files organization user may find when unzipping Generic Sensor Hub archive.

Assuming archive is unzipped in <target_dir> directory file tree is the following:

```
<target_dir>/invn/firmware/gsh/sdk/nucleo/win32/<version>/

- bin
  - linaro-cm4-nucleo
    - prebuilt firmware and prebuilt library (closed source)

- config
  - linaro-cm4-nucleo
    - main.mk
      - main makefile used to build the firmware out of any IDE
  - eclipse
    - ready to use Eclipse project
      - .project
      - .project
      - Debug-GSH_EXAMPLE-(load).launch
        - debug config to flash and attach to target
      - Debug-GSH_EXAMPLE-(no-load).launch
        - debug config to attach to target only
      - Makefile
      - template
        - contains a templated version of the Eclipse project (can be dynamically configured by a script to generate a Makefile for your project)

- doc
  - Dynamic_Protocol_Description.pdf
  - Generic_Sensor_Hub_Quick_Start_Guide.pdf
  - Generic_Sensor_Hub_Software_Guide.pdf

- html
  - doxygen documentation. Open index.html to open home page.

- sources
  - GSH
    - GshConfig.c
      - GshConfig.* aims at gathering all board and RTOS dependant code
    - GshConfig.h
    - GshFwVersion.h
      - just GSH software version number
    - main.c
      - just the main() function. Initialize the whole system.
    - stm32f411_flash.ld
      - linker file

- ExtFwk
  - examples
    - CustomSensor examples source code

- FreeRtos
  - FreeRTOSConfig.h
    - FreeRtos configuration file

- FrontEnd
  - GshDecimator.c
    - GshDecimator.h
    - FrontEnd is the interface of GSH with outside world. It also contains implementation of decimators.
```
VSensorImpl files are the implementation of the various virtual sensors embedded in GSH. They are compliant with VSensor framework. Thus they are able to listen some other Sensors (hardware or virtual) and notify their own set of listeners.

Such VSensors are initialized and registered at frontend level at boot time.

```
VSensorImplAkm09911.c : VSensor listening to Akm09911 magnetometer. Notifies raw data.
VSensorImplAkm09911.h
VSensorImplBmp280.c : VSensor listening to Bmp280 pressure and temperature sensor. Notifies raw data.
VSensorImplBmp280.h
VSensorImplIcm20690.c : VSensor listening to Invn Icm20690. Notifies accelerometer and gyroscope raw data.
VSensorImplIcm20690.h
VSensorImplVcnl4040.c : VSensor listening to Vcnl4040 proximity and light sensor.
VSensorImplVcnl4040.h
VSensorImplCModelAAAR.c : various VSensors. They all listen to one or more VSensor and may use some algorithms to notify elaborated data to their listeners.
VSensorImplCModelCalAcc.c
VSensorImplCModelCalGyr.c
VSensorImplCModelCalMag.c
VSensorImplCModelGravity.c
VSensorImplCModelGRV.c
VSensorImplCModelLinearAcc.c
VSensorImplCModelOrientation.c
VSensorImplCModelPickUp.c
VSensorImplCModelRV.c
VSensorImplCModel.h
VSensorImplCModelUtils.c : VSensor common functionalities
VSensorImplCModelUtils.h
```

Invn : contains all Invensense generic components may it be algorithms, VSensor framework, basic utilities, or communication protocol.

aar : Advanced Activity Recognition algorithms API

calibration : calibration algorithms API

common : basic common types definition

devices : hardware sensors abstraction layer

Drivers

Ak0991x
Bmp280
Icm20690
Vcnl4040

DynamicProtocol : Protocol used by GSH frontend to send/receive data to/from host

EmbUtils : basic functionalities implementation
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |

In this document the path invn/firmware/gsh/sdk/nucleo/win32/<version>/ is referenced as <gsh_root_dir>.
4 FREQUENTLY ASKED QUESTION

Next chapters give guidelines to realize the most common modifications of GSH in terms of configuration and extension.

4.1 HOW DO I ADD A NEW VSENSOR?

This chapter gives some guidelines to build a new VSensor. Even if it highlights some key points of this process, it does not remind all rules and restrictions imposed by VSensor Framework. Thus it is assumed that user will implement VSensor software in line with VSensor framework rules.

4.1.1 Create a new VSensor

Let’s take the example of a new VSensor named VSensorImplCMModelMyNewSensor based on existing VSensorImplCMModelGRV.c. Creation is achieved as follow.

In directory <gsh_root_dir>/sources/GSH/VSensorImpl:
- Copy-paste VSensorImplCMModelGRV.c into VSensorImplCMModelMyNewSensor.c
- Edit VSensorImplCMModelMyNewSensor.c
- This file starts with the VSensor class declaration sVSensorImplCMModelGRV. Let’s update it to:

```c
static struct {
    struct VSensor   vsensor;
    VSensorListener  listener;
    VSensorMyNewData data;
} sVSensorImplCMModelMyNewSensor;
```

**Note:** first field of VSensor class declaration must be of type struct VSensor to inherit from VSensor base class.

**Note:** in this example we choose to listen to only one VSensor. So there is only one VSensorListener field. It could be more as it is the case in VSensorImplCMModelGRV.c.

**Note:** our new VSensor handles a new specific data type that we call VSensorMyNewData. See §4.1.2 for data type declaration.

- Remove VSensorImplCMModelGRV_gyrEventHandler function because new VSensor has only one listener. Thus we need only one event handler.
- Rename functions with relevant prefix: VSensorImplCMModelGRV_* to VSensorImplCMModelMyNewVSensor_*
- Add exported functions prototypes to <gsh_root_dir>/sources/GSH/VSensorImpl/VSensorImplCMModel.h:

```c
void VSensorImplCMModelMyNewSensor_init(VSensor *sensorToBeListenedTo);
VSensor * VSensorImplCMModelMyNewSensor_getHandle(void);
```

**Note:** we have only one listener, that’s why init function only needs one VSensor pointer in parameter.

- Update code in functions:
  - VSensorImplCMModelMyNewVSensor_EventHandler
  - VSensorImplCMModelMyNewVSensor_update
  - VSensorImplCMModelMyNewVSensor_init
  - VSensorImplCMModelMyNewVSensor_getHandle

New VSensor is now created.

4.1.2 Add new VSensor type and data

We need to give our new VSensor a unique ID in VSensor framework and to declare new VSensor data type.
To declare a new specific type id edit <gsh_root_dir>/sources/Invn/VSensor/VSensorType.h and add a new specific macro declaration:

```c
#define VSENSOR_TYPE_MY_NEW_VSENSOR
```

To add new specific data type edit <gsh_root_dir>/sources/Invn/VSensor/VSensorData.h and add data structure declaration:

```c
typedef struct VSensorMyNewData {
    VSensorData base;
    /* use any type but make sure total size of following fields
     * does not exceed 64 bytes
     */
    uint32_t myNewSensorData;
    uint32_t myNewSensorData2;
} VSensorMyNewData;
```

**Note:** first field of VSensor data type declaration must be of type VSensorData to inherit from VSensor data base class.

### 4.1.3 Add support at Dynamic Protocol Adapter level

To have new VSensor supported at protocol level user need to do the following steps:

- Define a new Sensor type in enum DynSensorType in file <gsh_root_dir>/sources/GSH/FrontEnd/DynProtocol.h
- Add encoding and decoding support for this Sensor type in <gsh_root_dir>/sources/GSH/FrontEnd/DynProtocol.c
  - DynProtocol_getPrecision
  - DynProtocol_getPayload
  - DynProtocol_decodeSensorEvent
  - DynProtocol_encodeSensorEvent
  - DynProtocol_sensorTypeToStr

### 4.1.4 Plug VSensor at GshFrontEnd level

Here are the steps to have GshFrontEnd instantiating the new VSensor:

- edit <gsh_root_dir>/sources/GSH/FrontEnd/GshFrontEnd.c
- in function InvGshFrontEnd_initVsensorFlow()
  - add a new row to idata[] tab

```c
{ DYN_PRO_SENSOR_TYPE_NEW_SENSOR, VSensorImplCModelMyNewSensor_getHandle(),
  INV_GSH_FRONTEND_DECIMATOR_SIMPLE },
```

**Note:** user may choose among none, simple and advanced decimator as defined in GshFrontEnd.h.

  - add call to new Vensor init function

```c
VSensorImplCModelMyNewSensor_init(VSensorImplCModelCalAcc_getHandle());
```

**Note:** parameter is the VSensor object(s) that is (are) to be listened to. In this example the new VSensor listens to calibrated accelerometer.
4.2 HOW DO I ADD A NEW CUSTOM SENSOR?

Creation of a new custom sensor may be achieved in 3 steps:

- **Create MyNewCustomSensor.c file**
  
  In the source file add at least:
  
  - A static container for the custom sensor instantiation:
    
    ```c
    static inv_shext_vsensor_t MyNewCustomSensor;
    ```
  
  - The custom sensor event handler as a static function of type:
    
    ```c
    static int MyNewCustomSensor_eventCb(struct inv_shext_vsensor * sensor,
                        enum inv_shext_vsensor_event event, const void * data, uint16_t size)
    {
        // Event handler code here
    }
    ```
  
  - For more convenience we recommend to also implement a public function inside MyNewCustomSensor.c dedicated to sensor registering:
    
    ```c
    void MyNewCustomSensor_entry(void)
    {
        /* initialize custom sensor object and register it to the system */
        inv_shext_vsensor_register(&MyNewCustomSensor,
                         MyNewCustomSensor_eventCb, VSENSOR_TYPE_CUSTOM(0));
    }
    ```

  **Note**: 4 examples of custom sensor provided in `<gsh_root_dir>/sources/GSH/ExtFwk/examples` are a good start point to create a new custom sensor. See §2.3.4 for instructions on how to run these examples.

- **Declare custom source file in makefile**
  
  In `<gsh_root_dir>/config/linaro-cm4-nucleo/main.mk` add custom source file to CSRCS variable. For example:

  ```makefile
  CSRCS := \
  sources/GSH/ExtFwk/MyNewCustomSensor.c \n  ```

- **Register custom sensor in Generic Sensor Hub**
  
  To have the new custom sensor instantiated and run in GSH, user should call custom sensor registering function in `<gsh_root_dir>/sources/GSH/ExtFwk/SensorHubExt.c:inv_shext_entry()`.

  ```c
  extern void MyNewCustomSensor_entry(void);
  #pragma weak inv_shext_entry
  void inv_shext_entry(void)
  {
      MyCustomSensor0_entry();
  }
  ```

  **Note**: we recommend not to modify `<gsh_root_dir>/sources/GSH/ExtFwk/SensorHubExt.c`. That’s why function `inv_shext_entry` is declared as weak so that it may be overloaded. Thus a cleaner approach is to add a source file dedicated to custom sensors registering that overloads implementation of function `inv_shext_entry`. A such file exists as an example in `<gsh_root_dir>/sources/GSH/ExtFwk/examples/CustomEntry.c`
4.3 IS IT POSSIBLE TO USE A DIFFERENT HARDWARE SENSOR?

It is possible to add support for additional hardware sensors in GSH. Supporting a new hardware sensor requires to add two pieces of software:

- a hardware sensor driver that physically accesses and controls the sensor.
- a VSensor implementation that will use hardware driver to extract sensor data and make them available in VSensor framework.

It is a good practice to mimic the software of one of the currently supported hardware sensors. Sensors hardware drivers are located in `<gsh_root_dir>/sources/Invn/Devices/Drivers/<hardware-sensor>` while VSensor implementations are located in `<gsh_root_dir>/sources/GSH/VSensorImpl`. See §4.1 for details on how to add a new VSensor.

4.4 IS IT POSSIBLE TO USE A DIFFERENT MCU?

Generic Sensor Hub uses a Nucleo STM32F411RE as reference design. Anyway it is possible to use a different hardware platform. This chapter provides guidelines on how to move GSH from Nucleo to another platform.

Note: Generic Sensor Hub is expected to run on a cortex M4. Thus algorithms are provided in GSH in a library compiled for cortex-M4. In case user would target a different core but still use any VSensor depending on algorithm library, please contact your InvenSense representative.

4.4.1 Port board_hal

In GSH sdk, nucleo board is abstracted by a board_hal layer stored in `<gsh_root_dir>/sources/board_hal/`. Current board_hal is tightly linked to stm32f4xx drivers stored under `<gsh_root_dir>/sources/stm32f4x`.

To port GSH to another platform:

- Import targeted platform driver under `<gsh_root_dir>/sources/<target_driver>`
- port board_hal to the targeted platform drivers. Board_hal API should be kept unchanged.
- Update `<gsh_root_dir>/config/linaro-cm4-nucleo/main.mk` to remove all reference to STM drivers and add target platform driver

A few STM driver symbols are not abstracted by board_hal and thus directly used by GSH. User should pay particular attention to these symbols during porting effort. These symbols are listed in next chapters.

4.4.2 SystemCoreClock

SystemCoreClock is a global variable defined in `<gsh_root_dir>/sources/stm32f4x/CMSIS/Device/system_stm32f4xx.c`. It contains the system clock frequency.

It is used in `<gsh_root_dir>/sources/GSH/FreeRtos/FreeRTOSConfig.h`.

4.4.3 NVIC_SystemReset

NVIC_SystemReset executes a reset of the Nucleo board.

It is used in `<gsh_root_dir>/sources/GSH/main.c` in function `InvGshFrontEnd_executeSystemResetHook()`.

4.5 HOW DO I BUILD USER APPLICATION TASK?

A relevant use case is to run a full system on the MCU instead of limiting MCU to publish sensor data to a host. In that scope it is possible to have another RTOS task running in parallel of framework task. This task aims at running user application code and is able to send commands to framework task and receive sensors data from framework task.
This task may be activated at build time by setting the following macro in `<gsh_root_dir>/sources/GSH/GshConfig.c`:

```c
/** @brief Select the use of user_app thread
 *  1 : activate user app thread
 *  0 : user app thread is compiled out
 */
#define USE_USER_APP_THREAD 0
```

All code related to user application task is located in `<gsh_root_dir>/sources/GSH/GshConfig.c`. Once activated `<gsh_root_dir>/sources/GSH/GshConfig.c:user_app_thread()` runs the user application code (see §2.8.2 for more details on user application task).

### 4.6 IS IT POSSIBLE TO USE ANOTHER COMPILING SUITE?

It is possible to use GSH with a compiling tool suite other than linaro.

To do this user should execute the following steps:

- Add a dedicated directory under `<gsh_root_dir>/config` that will contain makefiles and project files if any.
- Make sure port files exists in FreeRTOS for the targeted compiling suite. Such files may be found in FreeRtos archive in `FreeRTOSV7.6.0\FreeRTOS\Source\portable`
- Pay attention to declaration of symbol `SystemCoreClock` in `<gsh_root_dir>/sources/FreeRTOS/FreeRTOSConfig.h`
## 5 DOCUMENT INFORMATION

### 5.1 REVISION HISTORY

<table>
<thead>
<tr>
<th>REVISION</th>
<th>DATE</th>
<th>DESCRIPTION</th>
<th>AUTHOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>August 11th, 2016</td>
<td>Initial version</td>
<td>Pierre PHILIPPE</td>
</tr>
<tr>
<td>1.1</td>
<td>October 11th, 2016</td>
<td>Align with GSHv1.1.0</td>
<td>Pierre PHILIPPE</td>
</tr>
</tbody>
</table>

Table 6. Revision History